TROUTE: A Reconfigurability-aware FPGA Router

Karel Bruneel and Dirk Stroobandt

Hardware and Embedded Systems Group, ELIS Dept., Ghent University,
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
{karel.bruneel;dirk.stroobandt}@UGent.be

Abstract. Since FPGAs are inherently reconfigurable, making FPGA designs generic does not reduce chip cost, as is the case for ASICs. However, designing and mapping lots of specialized FPGA designs introduces an extra EDA cost. We describe a two staged fully automatic FPGA tool flow that efficiently maps a generic HDL design to multiple specialized FPGA configurations. The mapping is fast enough to be executed online in dynamically reconfigurable systems. In this paper we focus on TROUTE, the routing algorithm used in our tool flow. We used TROUTE to implement reconfigurable Multistage Interconnection Networks and show huge improvements in area, speed and mapping time compared to conventional non-reconfigurable implementations.

1 Introduction

FPGA design differs significantly from ASIC design in the generality of the design solution. Indeed, ASIC designers need to amortize the NRE (non-recurring engineering) cost over a large volume of chip instances. This can be done by making the design more generic, so that it meets the needs of as many customers as possible. Besides the regular input data, a generic design takes *Parameter Data* as input, specifying how the regular input data should be processed. The configuration data can differ for each customer or group of customers and can even change over time. Naturally, more generic ASIC designs will be larger and possibly have somewhat lower performance, but the gains of selling more chip instances will in many cases outweigh these disadvantages.

FPGAs, on the other hand, are fully reconfigurable and therefore can be reused for any function of similar size and complexity. It is thus not useful to make an FPGA design as generic as possible because this will not make the chip any cheaper. On the contrary, you may need to switch to a more expensive FPGA to meet the area and performance cost of the generality. However, making lots of specialized designs now introduces an extra EDA cost as each specialized design must be designed separately and mapped to the FPGA. This is no longer feasible when we wish to switch between different designs at run time, in contrast to the generic ASIC solution.

Let us for example take the case of a communication network where each node has its own 128-bit encryption key. A generic ASIC design would store

the encryption key in an internal register. Each node can then be configured for a specific key by writing that register. In an FPGA we could use the same technique, but we could also save area and boost performance of the nodes by generating specialized FPGA bitstreams for each node. However, there are 2^{128} possible keys, making it infeasible to run the FPGA tool chain for each possible key. Also specializing the new configuration at run time for each key that is selected, is infeasible due to the amount of time it takes to synthesize such a new configuration.

To solve this problem we propose a two stage tool flow that drastically decreases the cost of generating a specialized FPGA configuration. The first stage of the tool flow takes a *Parameterizable HDL Design* as input and generates a *Parameterizable FPGA Configuration*. A parameterizable HDL design has two types of inputs: regular inputs and parameter inputs. The latter will not be inputs to the final design, but will be bounded to a constant value in the second stage (thus distinguishing between the various specialized configurations). In our encryption example, the key will be a parameter input. A parameterizable configuration is a set of Boolean functions that generates FPGA configurations given a parameter value. The second stage, generates specialized configurations by evaluating the parameterizable configuration for a parameter value. This can be repeated for multiple parameter values. One can easily see that for large numbers of parameter values the average cost per configuration is approximately the cost of running the second stage because the cost of generating the parameterizable configuration is amortized over all specialized configurations.

In [2, 3] we have shown that it is possible to build a two staged tool flow of which the evaluation of the parameterizable configuration runs 5 orders of magnitude faster than a conventional FPGA tool flow without sacrificing too much area and performance compared to a fully specialized FPGA design. In this tool flow only the LUT truth table bits of the configuration are expressed as Boolean functions of the parameters, whereas the routing is fixed for all configurations. In this paper, we present a tool flow that also expresses the routing configuration bits as a function of the parameter inputs. In the experiments section we show that this can result in a better area utilization and performance. Expressing the routing bits as Boolean functions of the parameters requires changes in all stages of the conventional FPGA tool flow (technology mapping, placement and routing). In this paper, we only discuss the changes in the router in detail. The other steps will be addressed only briefly.

2 Staged Mapping Tool Flow

Fig. 1 gives an overview of our mapping tool flow. The tool flow uses a compiler technique called staged compilation, or staged mapping, as we call it in the case of FPGA mapping. In our staged mapping flow the final result, a *Specialized FPGA configuration*, is generated in two steps or stages: the *Generic Stage* and the *Specialization Stage*. In contrast to conventional mapping the design specification is not entirely introduced at the start of the mapping process but a part of this

design specification is introduced at each stage. A Parameterizable HDL Design is introduced to the generic stage and the parameter values are introduced to the specialization stage. Each stage processes the result of the previous stage and the extra specification part to form a new intermediate result that will be introduced to the next stage. The generic stage produces a parameterizable configuration and the specialization stage combines this with the parameter values to produce the specialized configuration.

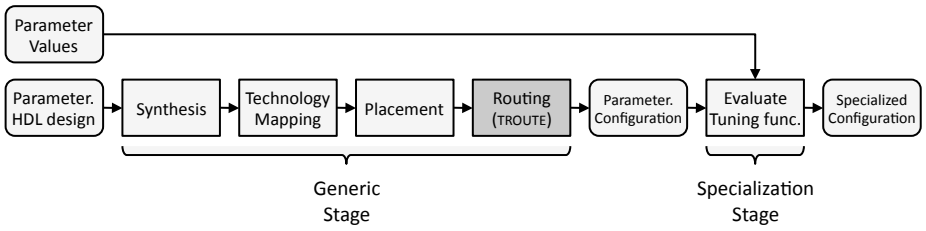

Fig. 1. Overview of our staged mapping tool flow.

A parameterizable configuration is a function that takes parameter values as arguments and produces a specialized configuration[1]. We represent a parameterizable configuration as a vector of Boolean functions whose elements are associated with the bits in the FPGA's configuration memory. The Boolean functions are called *Tuning Functions*. They are of closed form and have a single output.

The steps needed in the generic stage are similar to those used in conventional FPGA mapping: synthesis, technology mapping, place and route. In Section 3 we will explain these algorithms in more detail. It is important to note here that these algorithms are computationally hard and thus need a large run-time. The specialization stage generates a regular FPGA configuration by evaluation of the parameterizable configuration. This involves evaluating a set of closed form Boolean functions. Hence, the run-time of the second stage is linear in the size of the parameterizable configurations. The specialization stage will thus run a lot faster than the generic stage [2]. Therefore, the staged mapping tool flow is more efficient in generating specialized configurations than a conventional tool flow. This is because our staged flow can reuse the parameterizable configuration for each parameter value. The effort spend in the generic stage thus is divide over all invocations of the specialization stage. For large sets of parameter values the average mapping effort is approximately the effort pent in the specialization stage.

[1] The concept of parameterizable configurations can easily be extended to parameterizable partial configurations, which are functions that produce partial configurations when given the parameter values as argument.

3 Overview of the generic stage

The problem faced by the generic stage of our tool flow is to produce a parameterizable configuration given a parameterizable HDL description while optimizing some cost function. For the sake of clarity, we concentrate on minimizing the area used by the parameterizable configuration, but the techniques can be extended for other optimization criteria such as speed or a combination of area and speed. Without loss of generality, we will also assume a very simple island style target architecture with 4-input LUTs and wires of length 1.

Since both the input and output of the tool flow are parameterizable the internal data structures need to be able to express parameterizability and the algorithms that transform the data structures need to preserve the parameterizability. Similar to conventional FPGA mapping we divide the mapping problem into four subproblems: synthesis, technology mapping, placement and routing. In what follows we give an overview of these algorithms and the data structures used in our tool flow.

3.1 Synthesis

The synthesis step converts the parameterizable HDL description into a gate-level circuit. As we described in Section 2, a parameterizable HDL description distinguishes regular inputs from parameter inputs so, this distinction has to be preserved in the gate-level circuits. This can easily be done by allowing both types of inputs in the gate-level circuit data structure. The synthesis tool simply has to pass the information about the inputs.

3.2 Technology mapping

During technology mapping the gate-level circuit produced by the synthesis step is mapped on the resources available in the target FPGA architecture while trying to optimize the area of the implementation.

The result of a conventional technology mapper is a mapped circuit containing two types of functional blocks: LUTs and nets. A LUT can be implemented by a physical LUT on the FPGA and a net can be implemented by a subset of the routing switches in the configurable interconnect.

Because the bits in the parameterizable configuration are Boolean functions of the parameter inputs, both the truth table bits as well as the routing bits can change. First, a LUT with a truth table that is function of the parameters is called a *Tunable LUT (TLUT)* [2]. It's easy to see that a TLUT is a generalization of a regular LUT. Second, the way physical LUTs are connected can change depending on the parameters. We thus need functional blocks that reflect the parameterizability of interconnections. We call these blocks *Tunable Connections (TCONs)*. A circuit containing TLUTs (instead of regular LUTs) and TCONs (instead of nets) is called a *Tunable Circuit*.

A TCON has any number of input ports $I = \{i_0, i_1, \ldots, i_{L-1}\}$ and any number of output ports $O = \{o_0, o_1, \ldots, o_{M-1}\}$. Every TCON is associated to

a connection function f_{con} that shows how the output ports are connected to the input ports given a parameter value[2] $\boldsymbol{P} = (p_0, p_1, \ldots, p_{N-1}) \in \{0,1\}^N$, see equation (1). Just like a TLUT is a generalization of a regular LUT, it's easy to see that a TCON is a generalization of a net.

$$\begin{aligned} f_{con} : O \times \{0,1\}^N &\rightarrow I \\ (o, \boldsymbol{P}) &\mapsto i \end{aligned} \tag{1}$$

In what follows we will use a TCON with the functionality of a four-way switch, as example. This TCON has two inputs $\{i_0, i_1\}$ and two outputs $\{o_0, o_1\}$. The 1-bit parameter p controls how the inputs are connected to the outputs. When $p = 0$, o_0 is connected to i_0 and o_1 is connected to i_1. When $p = 1$, o_0 is connected to i_1 and o_1 is connected to i_0.

In this paper, we focus on the routing step of the tool flow. Therefore we assume the tunable circuit is given. More information on mapping to TLUTs can be found in [2, 3]. To our knowledge there are no technology mappers available that can map to a combination of TLUTs and TCONs.

3.3 Placement

During placement, each of the TLUTs in the tunable circuit is associated to (placed on) one of the physical LUTs of the FPGA while optimizing for a certain property, e.g. the routability of the placement.

Many FPGA placers use simulated annealing to place the mapped circuit. The cost function of a routability-driven placer is an estimate of the total number of wires the router will need to route the design given the current placement. This is calculated as the sum of the estimated number of wires used by the individual nets [1]. This same scheme is used to build a placer for tunable circuits. The only difference lies in the way we estimate the number of wires used by the router to route a TCON. In this paper we concentrate on the routing algorithm and therefore we assume the placement as given.

3.4 Routing

Conventional routers calculate the Boolean values that need to be stored in the configuration bits of the configurable interconnection network so that the physical LUTs are connected as is specified by the nets in the mapped circuit.

Our router is more complicated as it needs to calculate Boolean functions for the configuration bits. On one hand the parameterizable configuration evaluates to a specialized configuration given a parameter value and on the other hand a tunable circuit simplifies to a regular LUT circuit for that same parameter value. Our router will thus calculate Boolean functions for the routing bits so that for any parameter value the specialized configuration implements the connections specified by the regular LUT circuit.

In Section 4 we give a detailed description of an algorithm, called TROUTE that solves this problem.

[2] Without loss of generality, we combine all parameters into one parameter vector $\boldsymbol{P}$.

4 TROUTE

In this section we describe the algorithm TROUTE. Given a placed tunable circuit, it produces Boolean functions for the routing bits of the target FPGA so that the physical LUTs are connected as is specified by the TCONs of the tunable circuit. TROUTE is based on the widely used PATHFINDER algorithm [1, 7].

4.1 The Resource Graph

Both PATHFINDER and TROUTE uses a directed graph, called the *Resource Graph*, to represent the routing architecture of an FPGA. Because this graph can be constructed for many routing architectures, the algorithms are very flexible.

The resource graph is a directed graph $C = (V, E)$, where the vertices V represent the routing resources (the wires and the ports of the logic blocks). A directed edge (t, h) represents the possibility of routing a signal from resource t (the tail of the edge) to resource h (the head of the edge), by setting a switch. There are two types of port vertices: sources and sinks. Sources represent output ports of logic blocks while sinks represent input ports of logic blocks.

We can construct a resource graph as follows. Create a vertex v_r for each routing resource r (wire or port) of the target FPGA. For each unidirectional switch that, when closed, forces the logic value of resources i on resource o, create a directed edge (v_i, v_o), and for each bidirectional switch that connects resource r to resource s, create two directed edges (r, s) and (s, r). There are many extensions possible to this model [5] (beyond the scope of this paper).

Fig. 2 depicts the resource graph of a simple 2×2 island style FPGA with only length 1 wires and bidirectional switches. The wires are represented by solid black lines, the sinks and sources by small squares. The sinks are filled and the sources are not. For the sake of clarity we have not drawn all edges. The thin lines each represent two edges, one for each sense.

4.2 TCONs, patterns and nets

A TCON simplifies to a set of nets for a specific parameter value. We call this set of nets a *Connection Pattern* of the TCON. Each connection pattern describes one way to connect the output ports to the input ports of a TCON. A TCON can thus be represented as a set of connection patterns and a connection pattern as a set of nets.

When the placement of the LUTs is known the source vertex and the sink vertices associated to respectively the input port and output ports of the nets in the mapped circuit are known. Each net ν in the LUT circuit can thus be associated with an ordered pair (so_ν, SI_ν) containing a source vertex so_ν and a set of sinks vertices SI_ν. Note that due to the definition of a TCON (Section 3.2), the sink set of the nets in any pattern are disjoint.

A routing tree RT_ν for net ν is a rooted tree embedded in the resource graph C that has the source vertex as root and the sink vertices as its leaves. It does not contain any other source or sink vertices. This routing tree contains paths

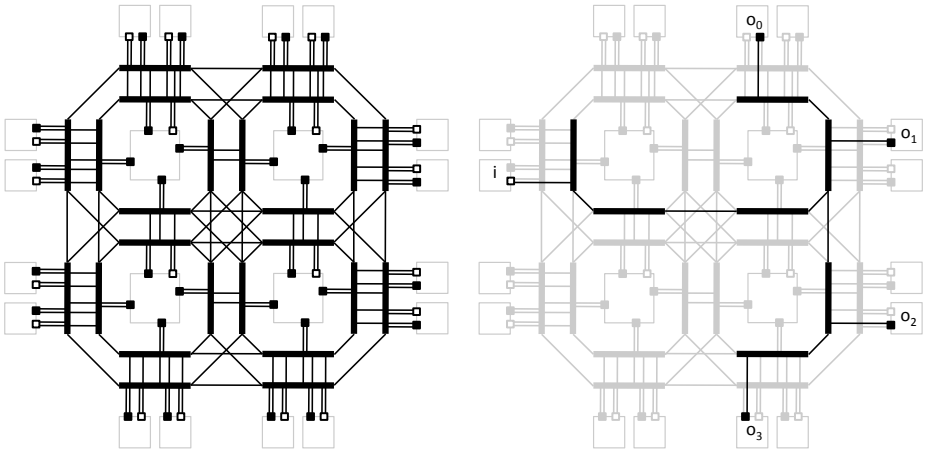

Fig. 2. (a) Resource graph for a simple 2×2 island style FPGA. Wires are solid black lines; Edges are thin lines; Sources are open boxes; And sinks are filled boxes. (b) Routing tree of a net $(i, \{o_0, o_1, o_2, o_3\})$.

from the source vertex to each sink vertex of the net ν. Fig. 2 shows the routing tree of a net $(i, \{o_0, o_1, o_2, o_3\})$. Once the routing tree RT_ν of a net is found, setting the FPGA's routing bits so all connections represented by the net ν are realized is easy. We just have to set the configuration bits so that the switches associated to the edges in RT_ν are closed, and the switches associated to the edges that only end or start in RT_ν are open.

Analog to the routing tree of a net, a routing graph RG_τ of a TCON τ is a subgraph embedded in the resource graph C. By controlling the switches associated to the edges in RG_τ it should be possible to realize all connections specified by the TCON. Therefore, RG_τ should contain a routing tree $RT_{\pi,\nu}$ for each net ν of each connection pattern π. Since the nets in a pattern coincide, their routing trees have to be disjoint. Nets that are part of different patterns, however, don't coincide. Their routing trees can therefore overlap. We define the routing graph RG_π of a pattern π as the union the routing trees $RT_{\pi,\nu}$.

Routing a tunable circuit thus simplifies to finding a set of disjoint routing graphs, one for each of the TCONs in the tunable circuit.

4.3 Tuning functions

Every connection pattern π can be associated to a Boolean function of the parameters, called the *Pattern Condition* $f^\pi_{cond}(\boldsymbol{P})$. This pattern condition is true for all parameter values that simplify the TCON to pattern π. Note that a TCON can simplify to the same pattern for several parameter values. Since every net is part of one pattern, a net is also associated to a pattern condition.

Once the routing tree $RT_{\pi,\nu}$ for each of the nets in the TCON routing graph RG_τ is found, the condition for a switch to close is given by the logical OR of all pattern conditions of those nets whose routing trees contain an edge associated to the switch. The tuning function for the configuration bit that controls the switch is equal to this condition or its inverse if the control of the switch is active high or active low respectively.

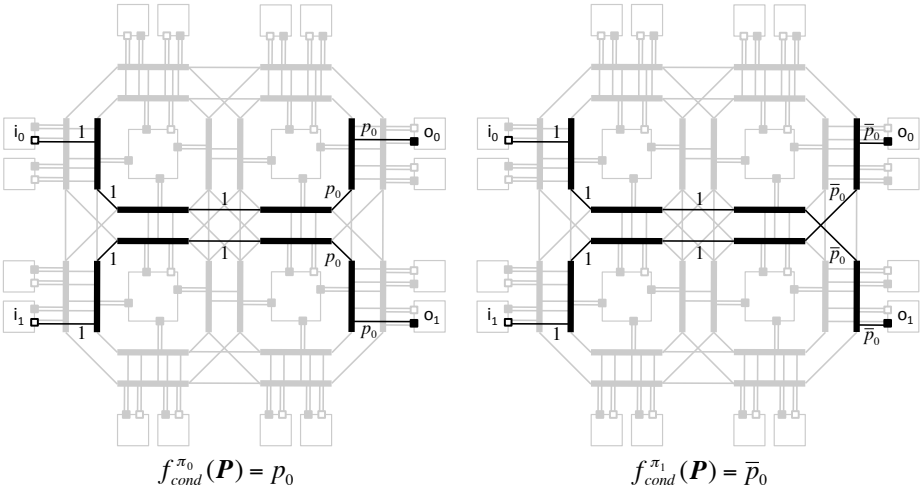

Fig. 3. Routing of the two connection patterns of a 4-way switch, their pattern conditions and the tuning functions.

A possible routing of our 4-way switch example is shown in Fig. 3. The figure on the left shows the routing trees of the two nets in pattern π_0 with condition $f^{\pi_0}_{cond}(\boldsymbol{P}) = p_0$ and the figure on the right shows the routing trees of the two nets in pattern π_1 with condition $f^{\pi_1}_{cond}(\boldsymbol{P}) = \overline{p_0}$. The routing graph of the TCON is the union of the routing trees. The edges that are crucial to the routing of the TCON are annotated with their tuning function (Fig. 3) assuming the switches are controlled active high.

4.4 The algorithm

The only problem left is finding a set of disjoint routing graphs, one for each of the TCONs in the tunable circuit.

First, we describe a heuristic subroutine that searches a minimum cost routing graph for a given TCON. Each vertex v in the resource graph has an associated cost c_v. The cost of a routing graph is the sum of the costs of its vertices. Second, we explain how to use this subroutine to find a set of disjoint routing graphs given a set of TCONs.

```
while shared resources exist:
  for each tcon τ do:
    τ.ripUpRouting()
    RG_τ = routeTcon(τ)
    for each vertex v in RG_τ:
      v.updateSharingCost()
  for each vertex v in C do
    v.updateHistoryCost()
```

Fig. 4. Main loop (Negotiated Congestion) of the TROUTE algorithm.

```
function routeTcon(tcon τ)
  RG_τ = null graph
  for each pattern π in tcon τ:
    RG_π = null graph
    for each net ν in pattern π:
      RT_ν = routeNet(net)
      for each vertex v in RT_ν:
        v.inPattern = true
        v.inTcon    = true
      RG_π = RG_π ∪ RT_ν
    for each vertex v in RG_π:
      v.inPattern = false
    RG_τ = RG_τ ∪ RG_π
  for each vertex v in RG_τ:
    v.inTcon = false
  return RG_τ
```

Fig. 5. Pseudo code for the TCON router.

The TCON router We will route a TCON by calculating a routing tree for each of the nets in the TCON. The union of all these routing trees is the routing graph of the TCON. We know that nets in a routing pattern coincide and thus have to be disjoint. However, two nets that are part of different patterns, never coincide and can thus share routing resources. We use this last property to minimize the routing cost of a TCON by maximizing the overlap among patterns.

The pseudo code of our proposed heuristic algorithm is shown in Fig. 5. The algorithm contains two nested for loops. The outer loop loops over all patterns of the TCON. The inner loop loops over all nets in the current pattern and routes them using a net router. A net router is a heuristic that searches a minimum cost routing tree for a given net. We use the net router described in [7].

In order to forbid resources sharing for nets within one pattern and allow resource sharing for nets in different patterns we manipulate the cost of the vertices within the TCON router. Therefore, we keep track of two extra flags for each vertex in the resource graph: *inTcon* and *inPattern*. The *inTcon* flag marks those resources that are used by already routed patterns of the TCON. The *inPattern* flag marks those resources that are used by already routed nets in the current pattern. These flags are used to calculate the manipulated cost of a resource c'_v, as is shown in equation 2.

$$c'_v = \begin{cases} \infty \text{ if } inPattern \\ 0 \quad \text{if } inTcon \wedge \overline{inPattern} \\ c_v \text{ otherwise} \end{cases} \tag{2}$$

There are three cases. The first case ensures that a resource that is already used in the current pattern cannot be used to route an other net in the current

pattern. The second case stimulates resource sharing when a resource is already in use by the TCON, but not by the current pattern. It does this by making the cost of these resources equal to zero. The third case is the default case.

Negotiated Congestion The TROUTE algorithm uses a mechanism called negotiated congestion to calculate a set of disjoint routing graphs for a given tunable circuit. The pseudo code of TROUTE is shown in Fig. 4. The algorithm iteratively rips up and reroutes (*routeTcon*) each of the TCONs until their routing graphs are disjoint. Or in other words, there are no shared resources.

In negotiated congestion, the individual routing problems are coupled by updating the vertex costs c_v during the routing process (*updateSharingCost* and *updateHistoryCost*). Our algorithm calculates and updates the vertex cost in exactly the same way as the routability-driven router described in [1].

5 Experiments and results

The TROUTE algorithm was implemented based on a Java version of the VPR (Versatile Place and Route) [1] routability-driven router, which we implemented. We use a simple FPGA architecture[3] with logic blocks containing one 4-LUT and one flip-flop. The wire segments in the interconnection network only span one logic block. The architecture is specified by three parameters: the number of logic element columns (*cols*), the number of logic element rows (*rows*) and the number of wires in a routing channel (W).

We validate TROUTE on Multistage Interconnect Networks that are known as Clos Networks [4]. Our Clos network uses 4×4 crossbar switches as building blocks. We use 4×4 switches because these can be efficiently implemented using four 4-input TLUTs or four TCONs. We compare three network types called: *Conv*, *Tlut* and *Tcon* each for three sizes 16×16 (3 stages), 64×64 (5 stages) and 256×256 (7 stages). *Conv* uses signals to control the crossbar switches while *Tlut* and *Tcon* use reconfiguration. *Tlut* only uses reconfiguration of LUT truth tables while *Tcon* uses both reconfiguration of LUTs and reconfiguration of routing. In *Tlut* all the switches are implemented with 4 TLUTs while in *Tcon* the switches in the even stages are implemented using TLUTs and the switches in the odd stages are implemented using TCONs.

We implemented the nine networks and measured: the number of LUTs, the number of wires, the logic depth, the routing time and the minimum channel width (W_m). Table 1 shows the results. The table also shows the parameters of the FPGA architecture. As suggested in [1], we ensure low-stress place and route by choosing the number of LUTs in the FPGA architecture 20% larger than the number of LUTs in the circuit and the number of wires per channel 20% larger than W_m, the minimum channel width.

[3] A description of this architecture is provided with the VPR tool suite in `4lut_sanitized.arch`.

Table 1. Properties of nine multi stage Clos network implementations. The numbers between brackets are relative compared to the *Tcon* implementation of the same size.

Impl.		Area		Speed	Routing		Architecture		
size	type	LUTs	wires	logic depth	$t_{route}[s]$	W_m	*cols*	*rows*	*W*
16	*Conv*	202 (12.63)	2131 (9.19)	5 (5.00)	7.96 (18.51)	6	20	20	7
	Tlut	48 (3.00)	526 (2.26)	3 (3.00)	1.06 (2.47)	4	10	10	5
	Tcon	16 (1.00)	232 (1.00)	1 (1.00)	0.43 (1.00)	5	8	8	6
64	*Conv*	1016 (7.94)	13613 (4.56)	9 (4.50)	294.73 (29.21)	6	47	47	7
	Tlut	320 (2.50)	3511 (1.17)	5 (2.50)	24.71 (2.45)	8	23	23	10
	Tcon	128 (1.00)	2987 (1.00)	2 (1.00)	10.09 (1.00)	9	18	18	11
256	*Conv*	6760 (8.80)	97994 (5.49)	12 (4.00)	15415.51 (25.09)	9	114	114	11
	Tlut	1792 (2.33)	25353 (1.42)	7 (2.33)	1234.66 (2.01)	13	53	53	16
	Tcon	768 (1.00)	17853 (1.00)	3 (1.00)	614.30 (1.00)	14	39	39	17

The wire utilization of the implementations will be influenced by the placement of the inputs of the network. If the inputs and outputs are placed far apart more wires will be needed than when they are placed close together. To normalize this influence we connect each input and output to a LUT that is connected to no other signals. This way the placer is free to place the inputs and outputs to minimize the number of wire resources. These extra LUTs are not accounted for in the LUT count of Table 1, because they are not part of the actual Clos network.

The routing of the *Conv* and *Tlut* implementations is done with the VPR routability-driven router. Their placement is done using the VPR routability-driven placer with default settings. The routing of the *Tcon* implementations is done using TROUTE. The placement is done using an adapted version of the VPR routability-driven placer, called TPLACE (beyond the scope of this paper).

The *Tcon* networks save up to a factor 8.8 in the number of LUTs compared to the *Conv* networks and up to a factor of 3 compared to the *Tlut* networks. As a measure for the clock speed we used the number of LUTs in the longest path (logic depth). When using the *Tcon* implementation, we can reduce the logic depth with up to a factor of 5 compared to the *Conv* implementation and a factor of 3 compared to the *Tlut* implementation.

The table also shows that up to a factor 5.49 can be saved in the number of wires compared to the *Conv* networks and up to a factor 2.26 compared to the *Tlut* networks. This last result might be counterintuitive since TCONs are more complex to route than nets. However, switching from *Conv* to *Tlut* to *Tcon* decreases the number of nets/TCONs and the number of LUTs. Less nets/TCONs connecting less LUTs that can be placed closer together thus results in less wires used. Because the LUTs get placed closer together W_m goes up, but it stays far from the channel widths used in commercial FPGAs.

Table 1 also shows the routing time needed for each implementation. All these experiments are done using an Intel Core 2 processor running at 2.13 GHz with 2 GiB of memory running the Java HotSpot^{TM} 64-Bit Server VM. Using

the *Tcon* networks we can save a factor of 18.51 up to 29.21 in the routing time compared to the *Conv* networks and a factor of 2.01 to 2.47 compared to the *Tlut* networks. This gain in routing time is due to the decrease in routing complexity as is explained in the previous paragraph.

6 Conclusions

In this paper we introduced a two staged FPGA tool flow that enables fast generation of FPGA configurations. The generic stage maps a parameterizable HDL design to a parameterizable configuration that expresses both the truth table bits and the routing bits as Boolean functions of the parameter inputs. We also provided a detailed description of TROUTE, the routing algorithm used in the generic stage of our tool flow.

We used TROUTE to implement reconfigurable Multistage Interconnection Networks similar to the implementations in [6, 8]. Since our design is done at the abstract level of tunable circuits, while theirs is done at the architectural level, our method greatly reduces the design effort. We have also shown that our implementations greatly improve area (LUTs: 8.80×, wires: 5.49×), logic depth (4×) and even routing time (25.09×) compared to a conventional non-reconfigurable implementation. These numbers are for a 256×256 Clos network.

References

1. V. Betz, J. Rose, and A. Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.
2. K. Bruneel and D. Stroobandt. Automatic generation of run-time parameterizable configurations. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. Kirchhoff Institute for Physics, 2008.
3. K. Bruneel and D. Stroobandt. Reconfigurability-aware structural mapping for LUT-based FPGAs. In *2008 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE, 2008.
4. C. Clos. A study of non-blocking switching networks. *The Bell System Technical Journal*, XXXII:406–424, 1953.
5. S. Hauck and A. Dehon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, November 2007.
6. P. Lysaght and D. Levi. Of gates and wires. *Parallel and Distributed Processing Symposium, International*, 4:132a, 2004.
7. L. McMurchie and C. Ebeling. Pathfinder: A negotiation-based performance-driven router for FPGAs. In *FPGA*, pages 111–117, 1995.
8. S. Young, P. Alfke, C. Fewer, S. McMillan, B. Blodget, and D. Levi. A high i/o reconfigurable crossbar switch. In *FCCM '03: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, 2003.